ROD Test Stand Software

Wisconsin Group

Khang Dao, Damon Fasching, Brian Holmes,

Richard Jared, John Joseph, Mark Nagel,

Sriram Sivasubramaniyan, Linda Stromburg

and Lukas Tomasek

Presented by Lukas Tomasek

(home institute: Institute of Physics AV CR, Prague)

Overview

• Test stand basic requirements and description

• Test stand software design

• Current software status, Host - Rod simulation

• Future plans

Basic requirements

- Short term:

  - initial testing and debugging the standalone ROD,
  - initial system test, BOC and TIM configuration and monitoring,

- Long term:

  - production testing,
  - ROD maintenance.

Test Stand hardware and software tools

- RODs, (TIM, ...),
- VME crate with NI VME-MXI-2 card,
- PC with NI PCI-MXI-2 interface card,
- MS Windows NT (2000) operating system,
- National Instruments LabWindows/CVI 5.5 development software,
- and **the application program - Host software**.

Test stand application program

- the programming language ANSI C,
- the programming style follows John Hill's "*Suggested Coding Rules*" common for both the DSP and the Host side:
  http://www-wisconsin.cern.ch/~atsiod/shared.html

- the code is shared in CVS repository (some header files common with DSP code).

The software design should be relatively independent on the platform, so the main parts of the the code could be used in the future version of the RCC.

Main application tasks:

- **similar to the future RCC**
- **ROD initialization and configuration:**
  - writing to RRFPGA flash memories and command registers, reading FPGA status registers;
  - loading program for DSPs;
  - BOC, TIM setup;

- **ROD control; communication/data exchange with Rod:**
  - "fast" *VmeCommandRegister* bit commands;
  - "**primitive**" commands => sending *PrimitiveLists* to Rod,
  - reading reply data (*ReplyLists*) from Rod,

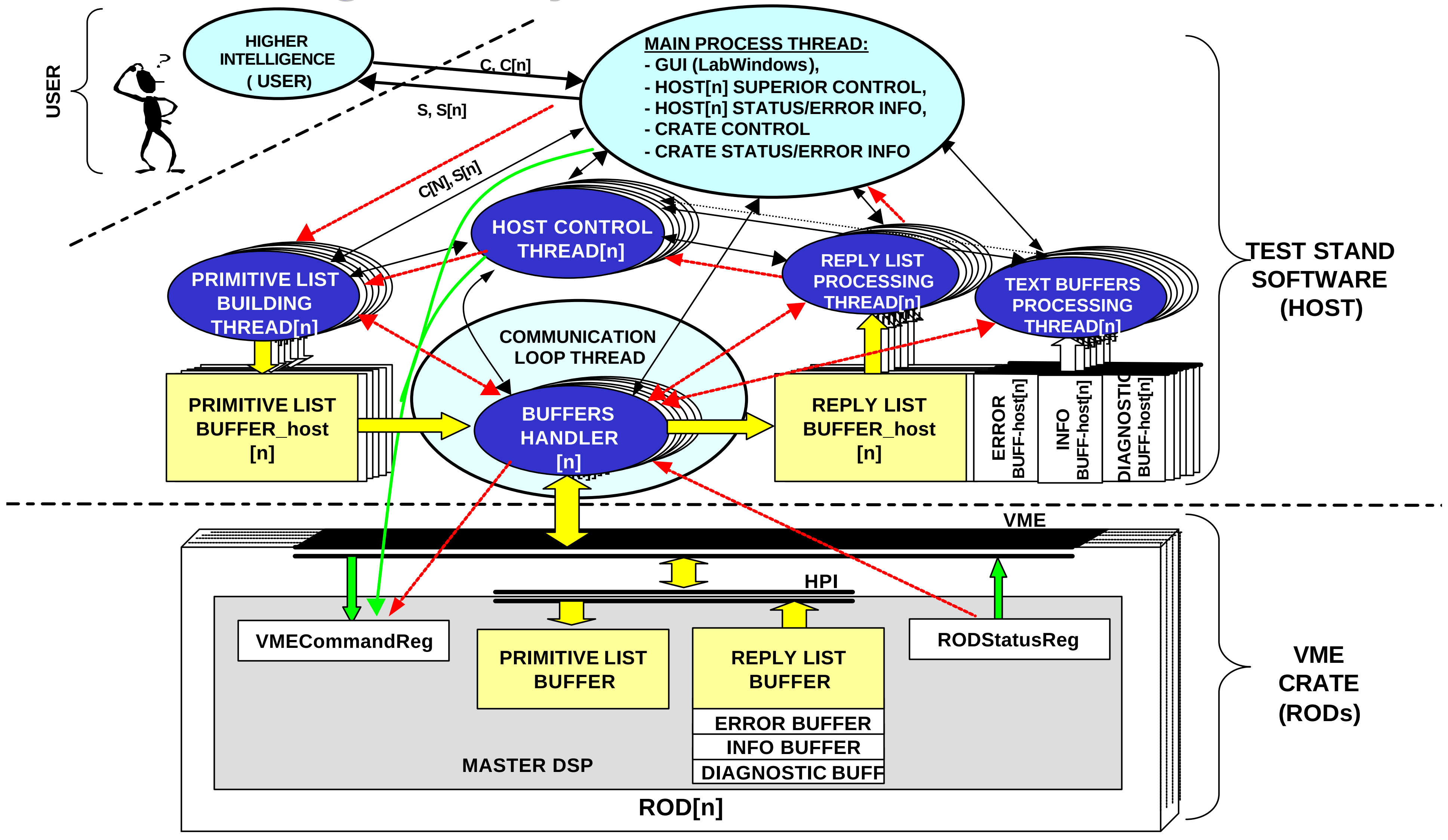
- **ROD status monitoring:**
  - periodic checking *RodStatusRegisters*,
  - readout error, info, diagnostic messages from the *TextBuffers*;

The primitives, input and output data formats, communication registers and communication scheme are described in the ***"Communication Protocol":***

http://www-wisconsin.cern.ch/~atsiod/shared.html

See also ***Damon Fasching's talk "Rod DSP Software"*** and RCC requirements in ***John Hill's talk "SCT ROD Crate DAQ".***

Host Program Layout and Host - Rod Interface



Legend:

	⇔	- Main data path
C	→	- Commands, control signals
S	←	- Status/error info
	↔	- Handshake/synchronization signals
	[n]	- Rod/Host index {0,1...15}

Note: FPGA registers accessed via VME omitted

Host - MasterDSP interface

- **communication registers (direct VME access):**

    VmeCommandRegs - the Host “fast” bit commands, the communication handshake bits; write/read access from Host, only read from MasterDSP.

    RodStatusRegs - status information, handshake bits; write/read from MasterDSP, read by Host.

- **data buffers (VME-HPI access):**

    PrimitiveListBuff - Host writes PrimLists, MasterDSP reads;

    ReplyListBuff - Master writes ReplyLists, Host reads.

- **text buffers (VME-HPI access):**

    Error, Info, Diagnostic Buffers (+corresponding textBuffStructs) - - written by DSPs, read by Host.

The VME-HPI interface is shown in the simulation section (Pg. 36).

Software design

- ***single process multithreaded*** application:

  - shared memory space -> simpler and faster synchronization and exchange of the information between the tasks than multiprocessing, doesn't have big demands on the system resources,

  - multithreading/multitasking is necessary mainly because of the need of a "responsive" user interface,

  - multitasking should not increase only the performance but also the "robustness" of the system by moving slow and potentially dangerous operations like file reading/writing outside the polling loop to the working threads;

  - the drawback is a difficult debugging, relatively complicated synchronization between the threads can lead to deadlock (if not done properly).

Thread hierarchy(from the top):

- ***User*** - superior to everything (human being!/or any higher level controller),
- ***MainProcessThread*** (highest, can control all threads),
- ***HostControlThread[n]*** (can control all threads with the same index, if it has a permission from MainThread),
- the rest are just ***"working" threads*** responsible for the low level data transfers to and from ROD driven by the synchronization signals.

Host thread priorities - basic “antagonistic” rules:

• ***Process priority class*** - normal
- PC should be able to run also other applications;

• ***All threads* =>** normal priority
- the Host software gets enough processor time slice by OS;

• ***MainProcessThread* > *the other threads* (golden rule)!!**
- the user interface must be responsive at any moment!!

• ***CommunicationThread* <= *the other threads !!***
- CommunicationLoopThread is still active (there are no additional delays inside), so it keeps the processor 100% busy. The other threads are active always only for a limited time -> can have a higher priority than CommLoop, but the “multitasking” effect is suppressed then.
If the working threads would have a lower priority there is a danger of lack of the processor time for them due to “frantically” running CommunicationLoop -> deadlock.

->the compromise - “default” values: MainThread one(or two) points above the rest, which are all on the same - normal - level.

Basic synchronization mechanisms between the threads used in the TestStand code:

• ***Win32 event*** - SetEvent(), ResetEvent(),
WaitForEvent() - sleep function, the thread is suspended until receives the specific event,
- used mainly for inter-thread synchronization;

• ***Win32 CriticalSection*** - protects the shared resources (VME bus, common files, e.g. common program error file) from a simultaneous multiple access so maximally one thread have a “key” to the specific shared parts of the program;

• ***Win32 SuspendThread() and ResumeThread(***)
- could be used for some simple synchronization tasks instead of the pair SetEvent() - WaitForEvent();

• ***shared volatile variable***
- universal, platform independent, easily readable;
- used usually as a status “flag”, often in the pair with one of the more sophisticated sync. mechanisms above.

Host Error Handling

• similar to the DSP code (Coding rules):

- every routine returns a status (except low level routines, where an error is “impossible”);
- if an error is detected the error handling routine is called:
  - host/rod[n] specific,
  - general program (low level routines);

```
/* host specific error routine prototype */
void hostError (const char* fileName, int line, ERROR_ID errorId, struct HOST *host, const char *errorMessage);

/* general program error routine prototype */
void programError(const char* fileName, int line, ERROR_ID errorId, const char *functionName, int functionStatus,
                                    const char *errorMessage);
```

Input parameters of the error handlig routines are:

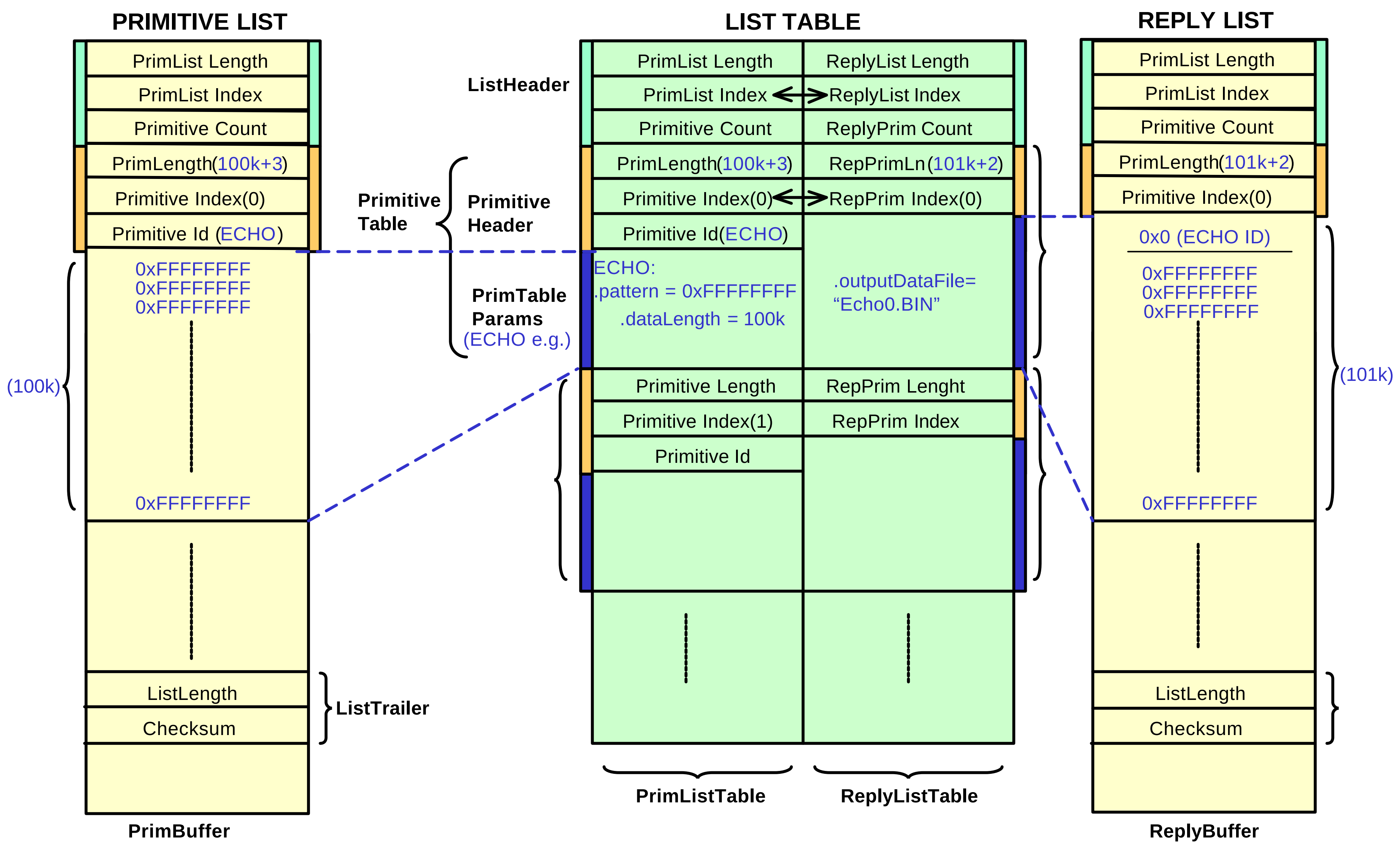
- source file name (C macro __FILE__) - mandatory,
- line number in this file (C macro __LINE__) - mandatory,
- errorId (the error code assigned by host) - required,
- functionName (the name of the function - if any - which returned the error);
- status - the function return code (if any);
- pointer to the host[n] struct which contains the status info about the host/rod[n] -
  - required in hostError();
- optional errorMessage.

"C" macros __FILE__ and __LINE__ allow a simple tracking of any nested error from the source to the end.

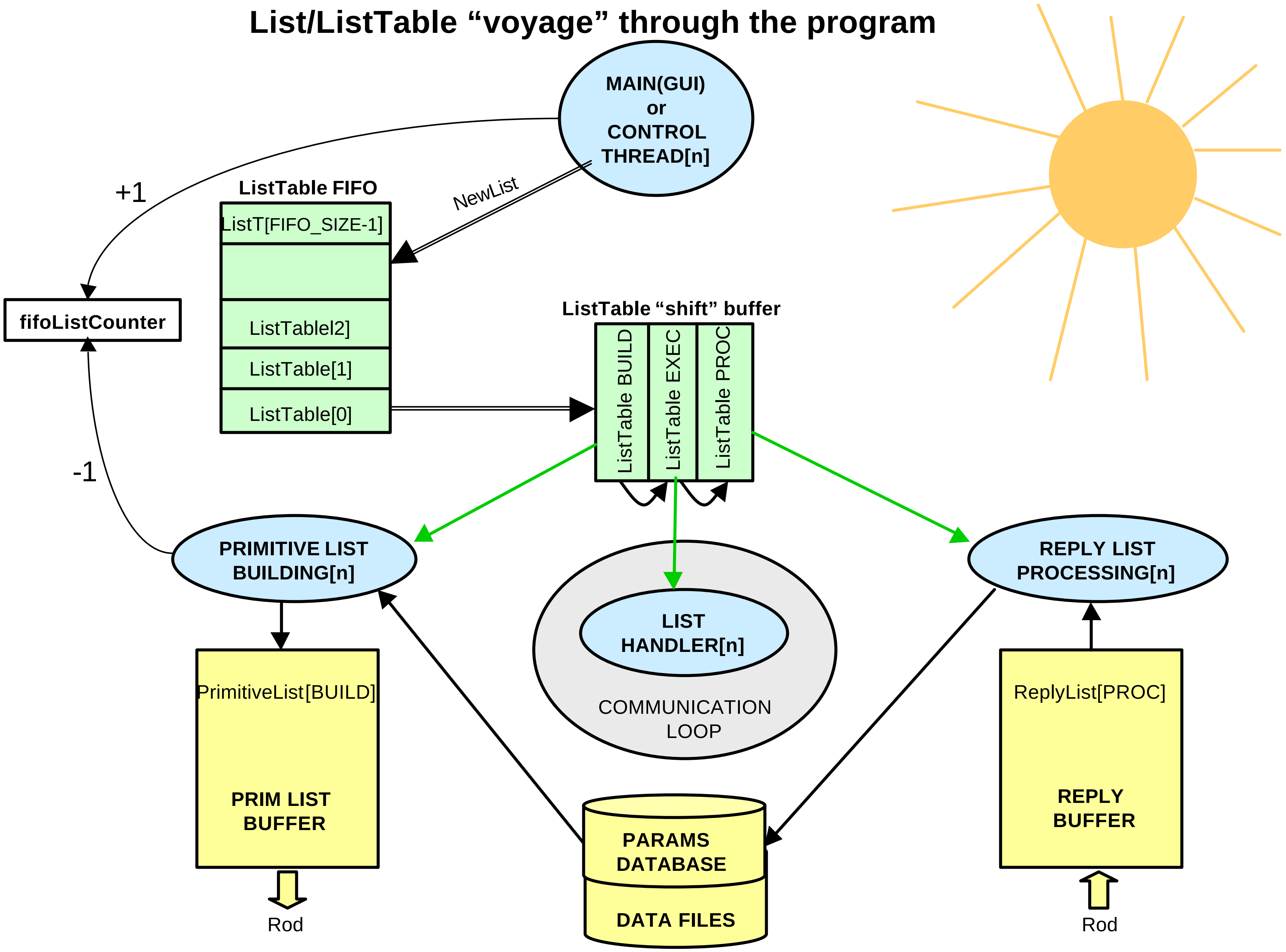
Both the general program and the host error routines append the error information to the common error file and also display on the screen.
No special error handling algorithms depending on the error ID are currently intended inside the error handling routines for the initial testing.
The decision what to do in the case of error (fatal -> abortive return from the function or move to suspended error state X non fatal -> continue) is done directly on the place where the error was detected).

More sophisticated error codes and error handling will be introduced in the future.

PrimitiveList - ListTable - ReplyList



List/ListTable “voyage” through the program



The commands for the Rods issued by User from MainThread(GUI) or programatically from ControlThread in the form of the ***ListTable*** are stored in ***ListTableFifo.***
ListTable struct - a sort of dynamic database used for ***PrimitiveList*** "prebuilding", building and processing its reply data. Length of the ListTable is static (limited number of primitives in the list) -> no dynamic memory allocation during runtime is necessary.
The parameters stored in ListTable together with the parameters stored in the common database or files contain all information necessary for unique "coding" of each primitive to ***PrimitiveList*** and "decoding" its reply data from ***ReplyList***. The main advantage - ListTable struct should be more "compact" than the corresponding PrimitiveList.

ListTableFifo used for a temporary storage the ListTables waiting for the execution is implemented as a circular buffer with the constant size -> static memory allocation.

ListTableShiftBuffer - the Host must remember information up to last three lists, since at the same moment one PrimList could be built in the host PrimBuffer (info stored in *ListTableBuild*), the second list already transferred to the Rod is executed by MasterDSP (info in *ListTableExec*) and the last one - ReplyList from the Host ReplyBuffer - is processed (info in *ListTableProcess*).

Primitive Function

- similar usage to DSP Code (see *Damon's talk "DSP Software"*).
The exact algorithm for the primitive ”prebuilding” into *ListTable*, building into *PrimList* and reply data processing from *ReplyList* is coded in the primitive function dedicated to each primitive (resp. primitive ID). The pointer to this function is stored in the common "array of pointers to the primitive functions":

```
 /* array of pointers to prim. functions declaration */
 ERROR_CODE (*primitiveFunction[NUMBER_OF_PRIMITIVES])(InputParamsDeclaration);

/* primitive function declarations */
 ERROR_CODE primFunc_echo[ECHO](InputParamsDeclaration);
 ERROR_CODE primFunc_memTest[MEM_TEST](InputParamsDeclaration);

/* array initialization */
 primitiveFunction[ECHO]=&primFunc_echo;
 primitiveFunction[MEM_TEST]=&primFunc_memTest;

/* primitive function calling example */
 int primitiveId=ECHO;
 errorCode=(*primitiveFunction[primitiveId])(inputParams);
```

The index of each primitive function in the array is equal to its primitiveID, which are defined “by name” in the header file common for both the Host and the DSPs.
The advantage - instead of long switch structures, any primitive function is called simply by primitiveID.

MainProcessThread

- main program thread - creates the other threads,
- main task - **interaction with the User** (LabWindows GUI),
  - controls the Host process,
  - sends commands for the Rod in the form of ListTable to ListTableFifo;
  - displays the Rods and Host process status information;
- "fast" commands - bits in VmeCommandRegs are sent directly from Main Thread, which has a higher priority than CommunicationThread -> VME access guaranteed asap;
- executes some single tasks (which are fast and “safe”) - GUI must not be frozen.

HostControlThread[n]

- controls and coordinates the execution of time consuming commands and procedures specific for Rod[n]/Host[n], which cannot be done in MainThread (for example ROD initialization, standard testing routines …).

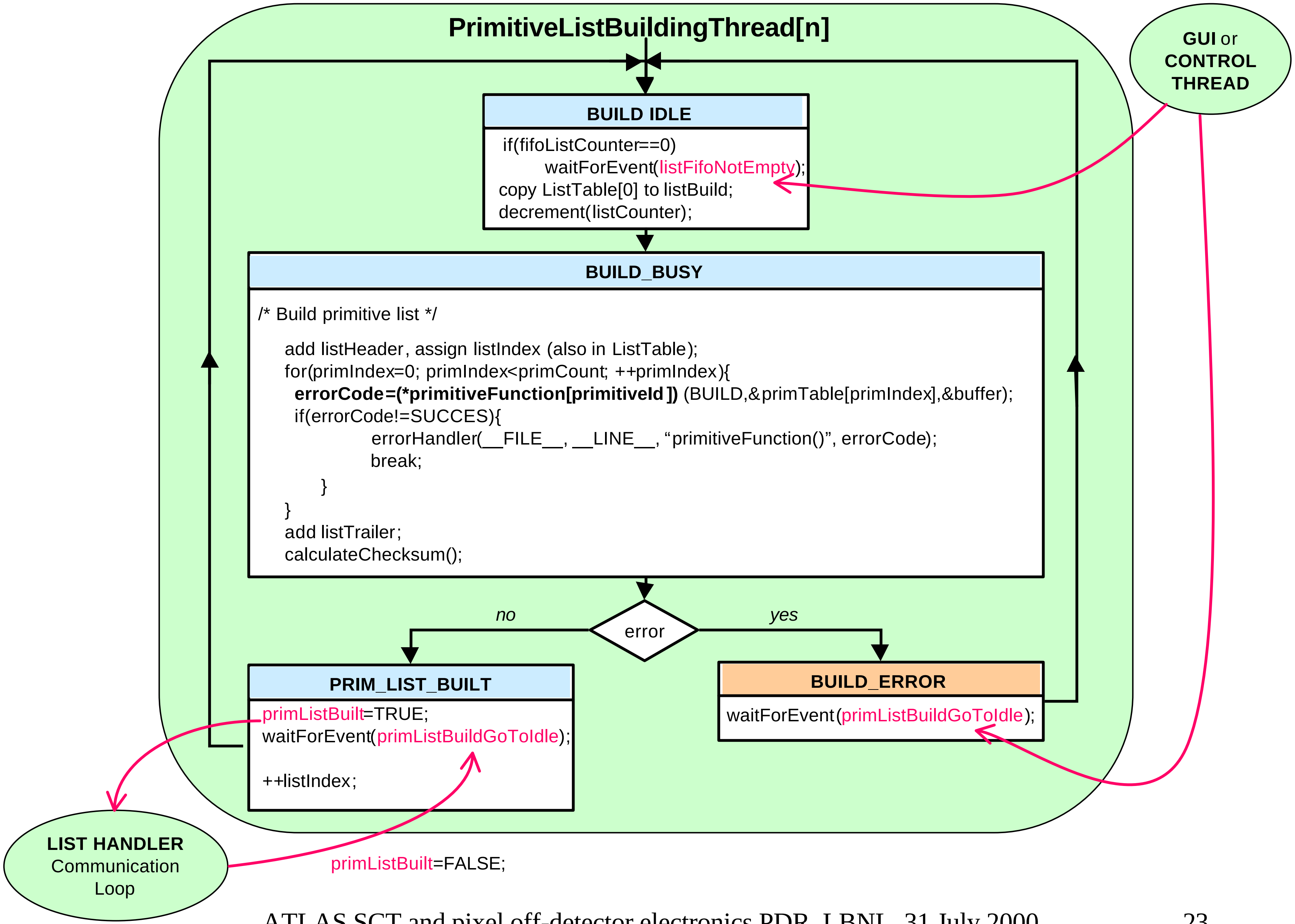
HostControlThread[n] - structure (pseudocode):

```
ERROR_CODE hostControl(uint rodNumber){

    while(programExit==FALSE) {
            /* thread suspended */
            SuspendCurrentThread();     /* waitForEvent(commandReady); */

            switch(command){

              case ROD_INIT:
                   /* load DSP software, wait for InitBit … */
                   errorCode=rodInit(hostNumber);
                   if(errorCode!=SUCCES){
                           errorHandler(__FILE__, __LINE__, "rodInit()", errorCode);
                   }
                   break;
              case TEST_A:         /* example */
                   errorCode=testA(rodNumber);  /* Rod test A */
                   if(errorCode!=SUCCES){
                           errorHandler(__FILE__, __LINE__, "testA()", errorCode);
                   }
                   break;
              case TEST_B:
                  .
                  etc.
            }
            controlThreadBusy=FALSE;
    }
    return(SUCCESS);
}
```

MainThread (GUI)
ResumeThread(hostControl);
command= ROD_INIT (e.g.);
controlThreadBusy=TRUE;

PrimitiveListBuildingThread[n]

• “builds” *PrimitiveList* in the Host local mirror of the *PrimitiveListBuffer* according to the commands issued by Main or Control thread and stored in the form of ListTable in ListTableFifo.



```
if(fifoListCounter==0)
        waitForEvent(listFifoNotEmpty);
copy ListTable[0] to listBuild;
decrement(listCounter);
```

```
/* Build primitive list */

     add listHeader, assign listIndex (also in ListTable);
     for(primIndex=0; primIndex<primCount; ++primIndex){
       errorCode=(*primitiveFunction[primitiveId]) (BUILD,&primTable[primIndex],&buffer);
       if(errorCode!=SUCCES){
                    errorHandler(__FILE__, __LINE__, "primitiveFunction()", errorCode);
                    break;
           }
     }
     add listTrailer;
     calculateChecksum();
```

```
primListBuilt=TRUE;
waitForEvent(primListBuildGoToIdle);

++listIndex;
```

```
waitForEvent(primListBuildGoToIdle);
```

```
primListBuilt=FALSE;
```

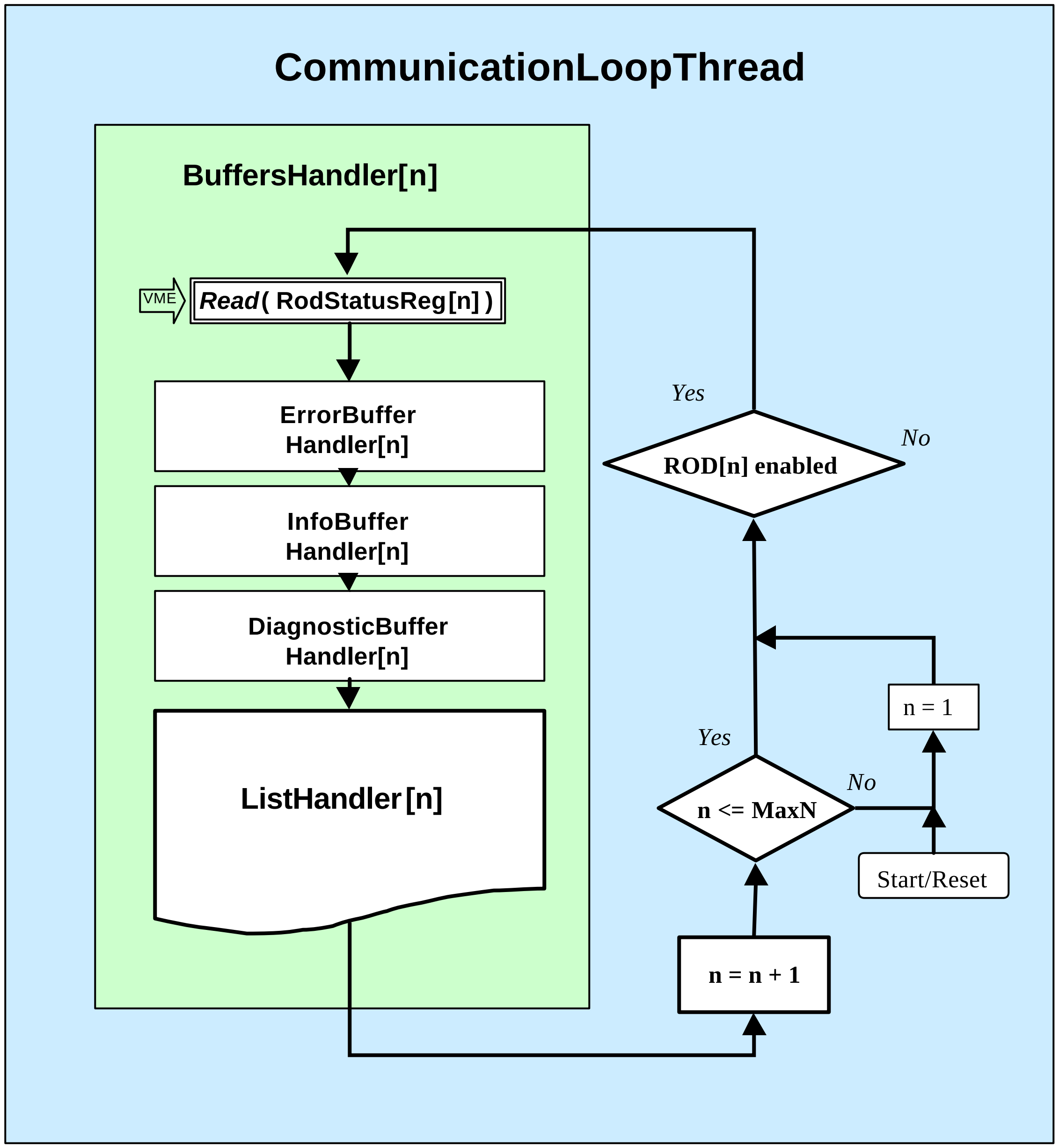
CommonCommunicationLoopThread - “polling loop”

• grants and distributes the access to the shared VME bus between all RODs in the crate on the ***“round robin”*** basis,

• periodically monitors the status of all Rods in the crate(RODStatusRegister),

• contains ListHandler and textBufferHandlers (error, info, diagnostic) routines which are responsible for the data transfers from and to MasterDSP.

The proper working all of these tasks is absolutely essential for the TestStand software functionality => must be extremely robust - should run “forever”.

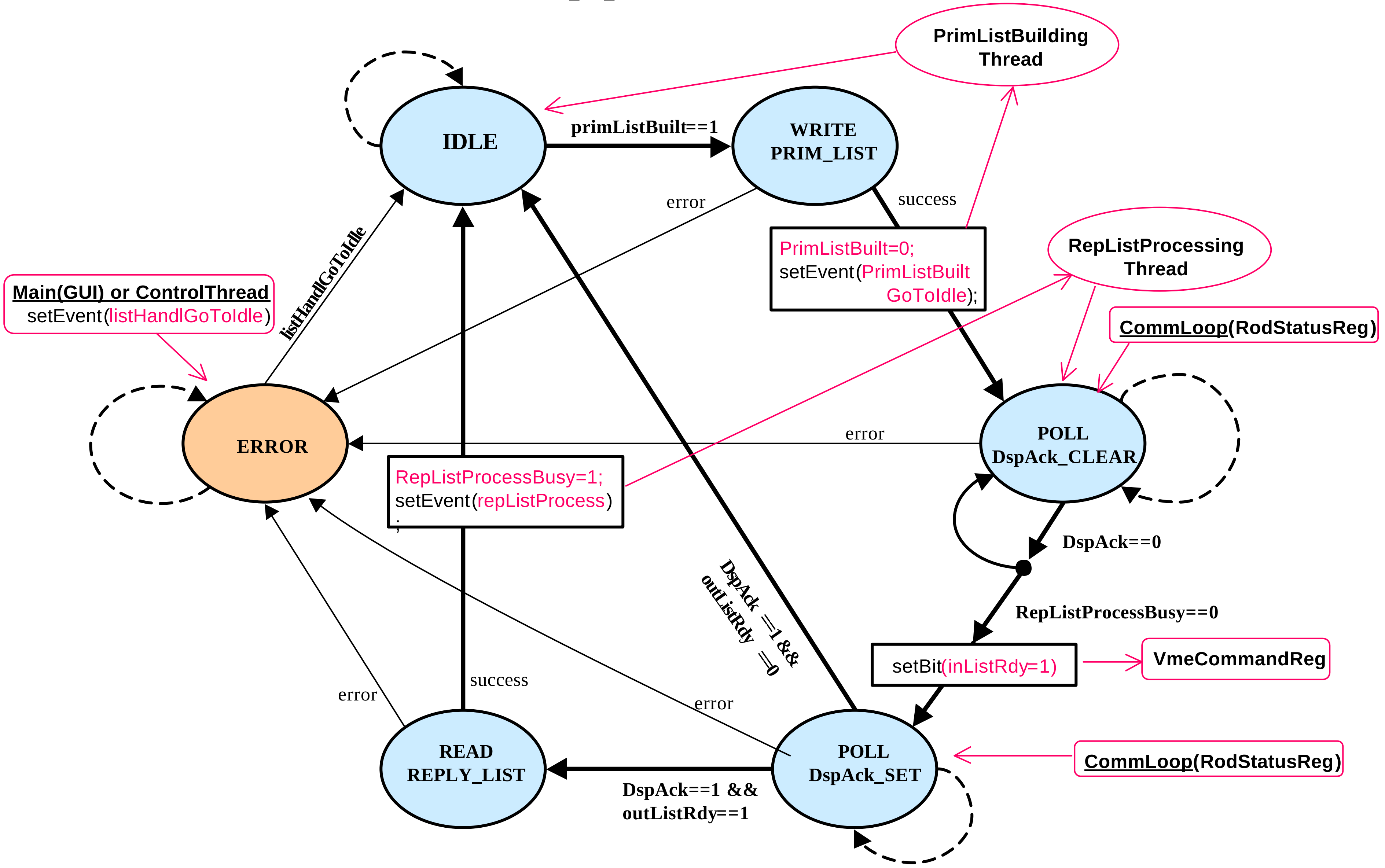

CommunicationLoopThread
BuffersHandler[n]
VME
Read(RodStatusReg[n])
ErrorBuffer Handler[n]
InfoBuffer Handler[n]
DiagnosticBuffer Handler[n]
ListHandler[n]
Yes
No
ROD[n] enabled
n = 1
Yes
No
n <= MaxN
Start/Reset
n = n + 1

ListHandler[n] routine

• data transfer and handshaking between ROD[n] and Host[n],
i.e. moving ***PrimitiveLists*** to MasterDSP (PrimListBuffer) and ***ReplyLists*** back (ReplyListBuffer);
- synchronization with MasterDSP and the list building and processing threads.

• The Host implementation of the***"Communication protocol"*** *is* a state machine.

ListHandler[n] - state machine



TextBufferHandler[n] routine (error, info and diagnostic)

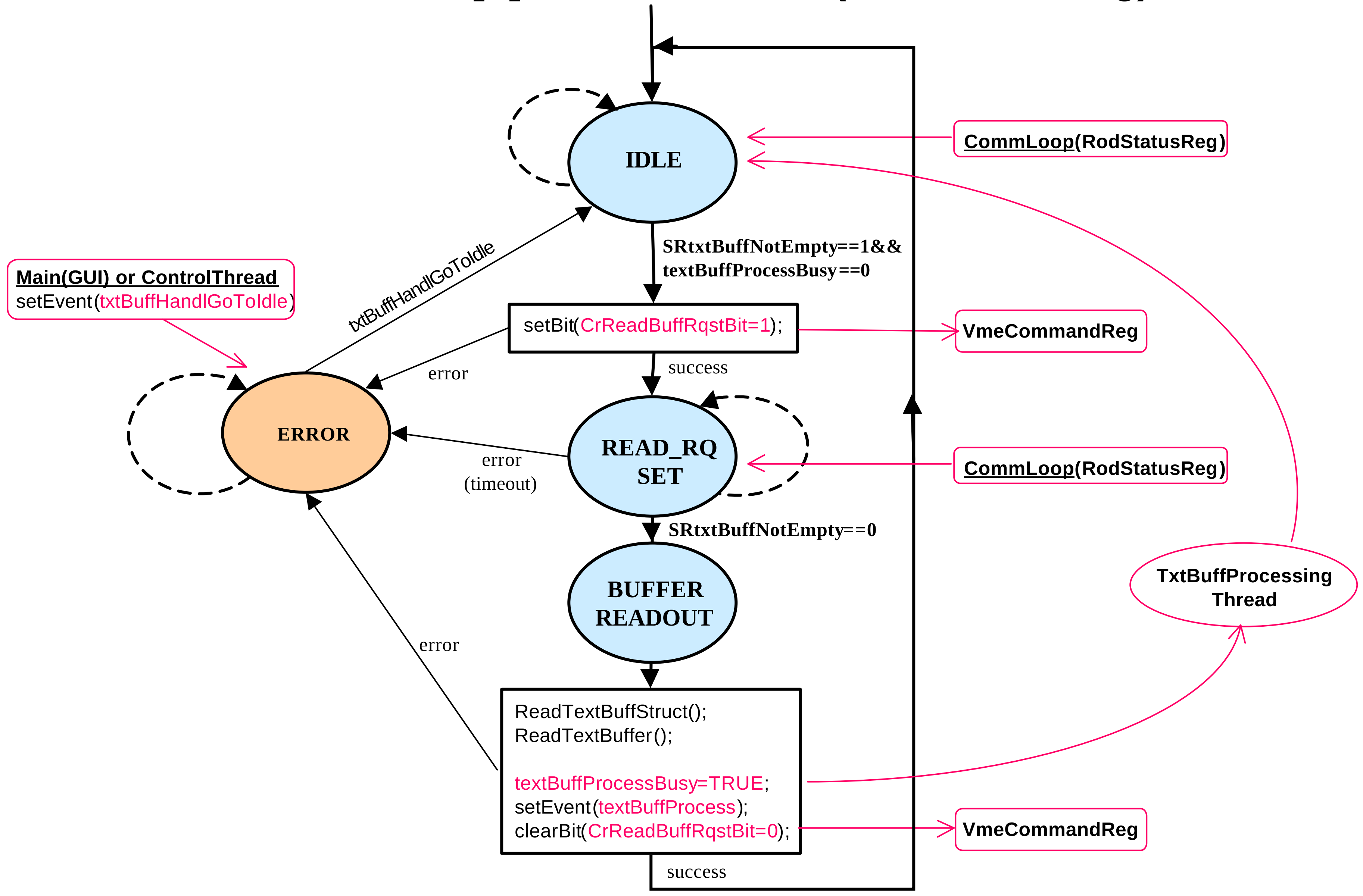
• readout text (error, info, diagnostic) messages from MasterDSP via VME, i.e. moving data from the MasterDSP textBuffers to the Host local textBuffers + synchronization handshaking with the Master;

- the current status information about the text buffer (data start, data end, mode, overflow flag) is stored in the MasterDSP memory location different from the corresponding textBuffer, so this TextBufferInfoStruct must be always read before each message readout;
- addresses of these textBufferInfoStructs are passed into the ReplyBuffer immediately after the MasterDSP initialization and consequently read by Host.

• when the message transfer is done, the textBuffProcessingThread is asked for the message "processing";

• implementation - state machine:

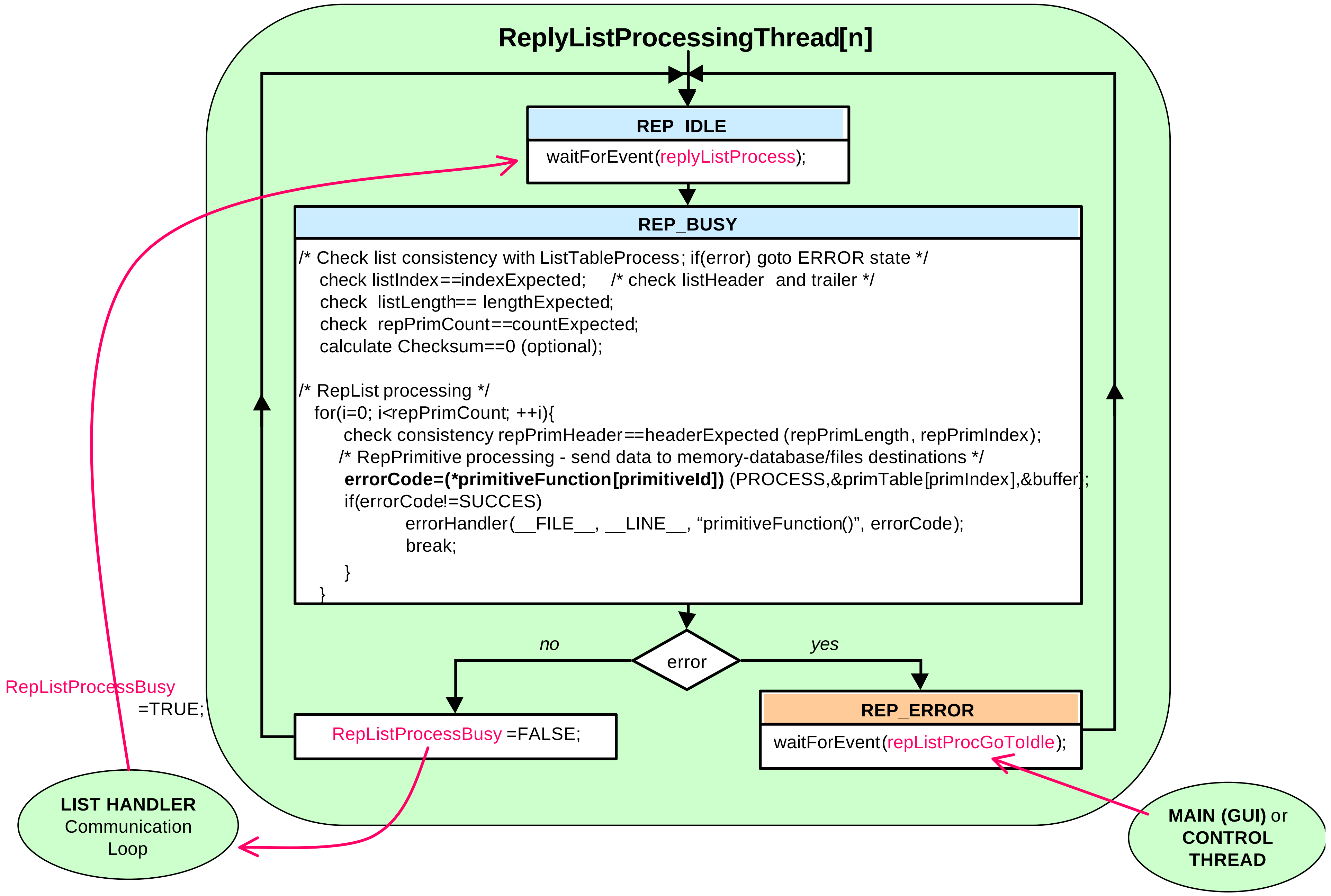
TextBufferHandler[n] - state machine (error, info, diag)



ReplyListProcessingThread[n]

- processes **ReplyList** stored in the local buffer, i.e. checks the validity of the data (checksum, length and consistency with the corresponding ListTableReply resp. PrimitiveList etc.) and then sends out the data to the predefined destinations (memory locations/database, files ...),

- common *primitiveFunction* (resp. dereference of the pointer to the primFunc) is used for the “decoding” of each “ReplyPrimitive” from ReplyList.


ReplyListProcessingThread[n]
REP_IDLE
REP_BUSY
no
error
yes
REP_ERROR
RepListProcessBusy
=TRUE;
LIST HANDLER Communication Loop
MAIN (GUI) or CONTROL THREAD

```
waitForEvent(replyListProcess);
```

```
/* Check list consistency with ListTableProcess; if(error) goto ERROR state */
    check listIndex==indexExpected;     /* check listHeader  and trailer */
    check  listLength== lengthExpected;
    check  repPrimCount==countExpected;
    calculate Checksum==0 (optional);

/* RepList processing */
   for(i=0; i<repPrimCount; ++i){
        check consistency repPrimHeader==headerExpected (repPrimLength, repPrimIndex);
       /* RepPrimitive processing - send data to memory-database/files destinations */
        errorCode=(*primitiveFunction[primitiveId]) (PROCESS,&primTable[primIndex],&buffer);
        if(errorCode!=SUCCES)
                    errorHandler(__FILE__, __LINE__, "primitiveFunction()", errorCode);
                    break;
        }
    }
```

```
RepListProcessBusy =FALSE;
```

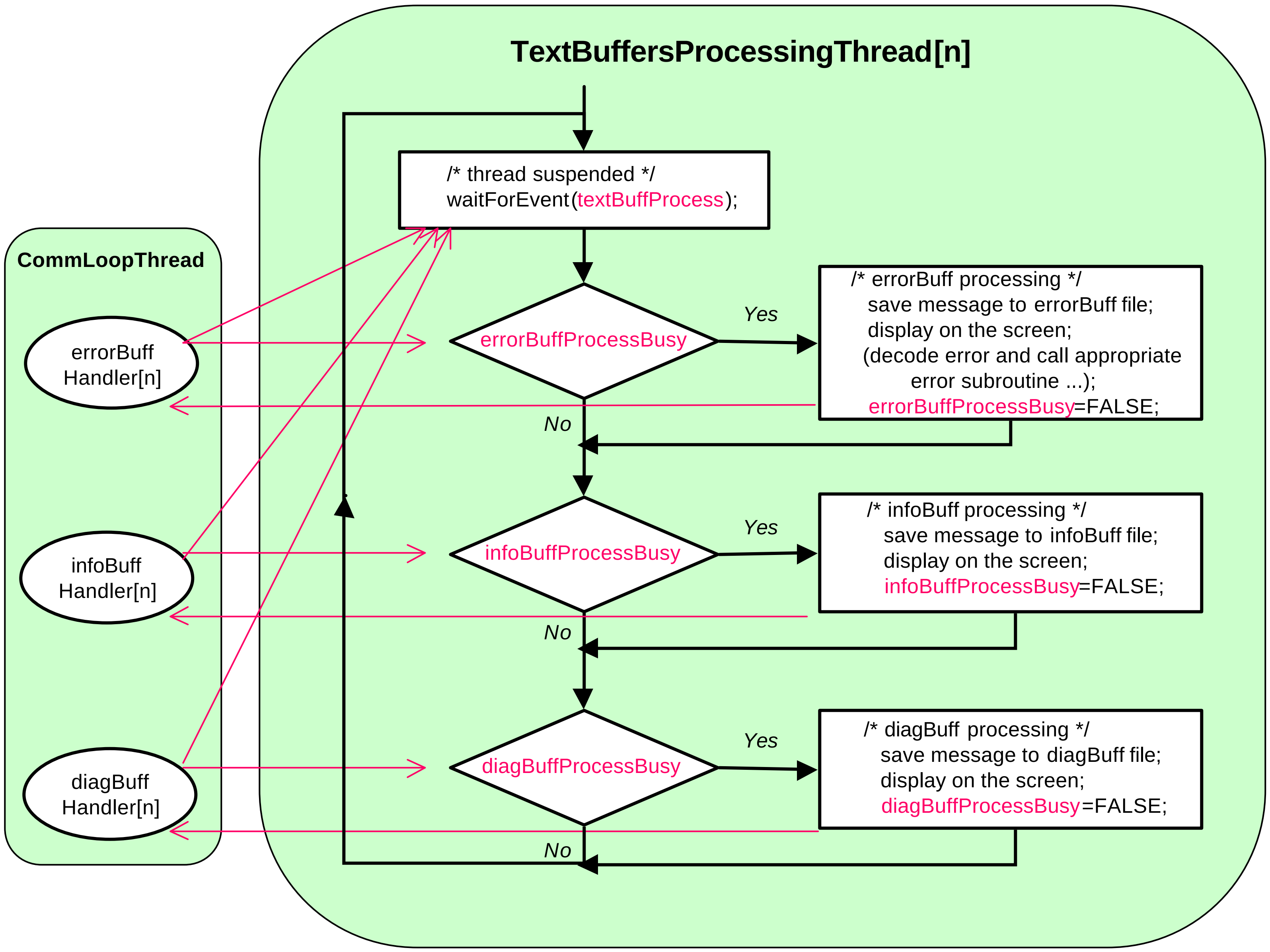
```
waitForEvent(repListProcGoToIdle);
```

TextBuffersProcessingThread[n]

• processes text messages (print to screen, save to file ...)
generated by Rod[n] and stored in the host local copies of
the error, info and diagnostic buffers;

• one common processing thread for all text buffers is fairly sufficient,
since the text buffers are and probably will be relatively small ->
-> message processing shouldn’t take long time and
we don’t expect a traffic jam on these buffers, do we?

• more sophisticated “decoding” of the Rod error messages
and the consecutive actions depending on the passed ErrorCode
will be implemented in the future.


TextBuffersProcessingThread[n]
/* thread suspended */
waitForEvent(textBuffProcess);
errorBuffProcessBusy
Yes
No
/* errorBuff processing */
save message to errorBuff file;
display on the screen;
(decode error and call appropriate
error subroutine ...);
errorBuffProcessBusy=FALSE;
infoBuffProcessBusy
Yes
No
/* infoBuff processing */
save message to infoBuff file;
display on the screen;
infoBuffProcessBusy=FALSE;
diagBuffProcessBusy
Yes
No
/* diagBuff processing */
save message to diagBuff file;
display on the screen;
diagBuffProcessBusy=FALSE;
CommLoopThread
errorBuff Handler[n]
infoBuff Handler[n]
diagBuff Handler[n]

Current software status

Has been done:

•"first iteration" of the Test Stand program has been designed and all major parts implemented:

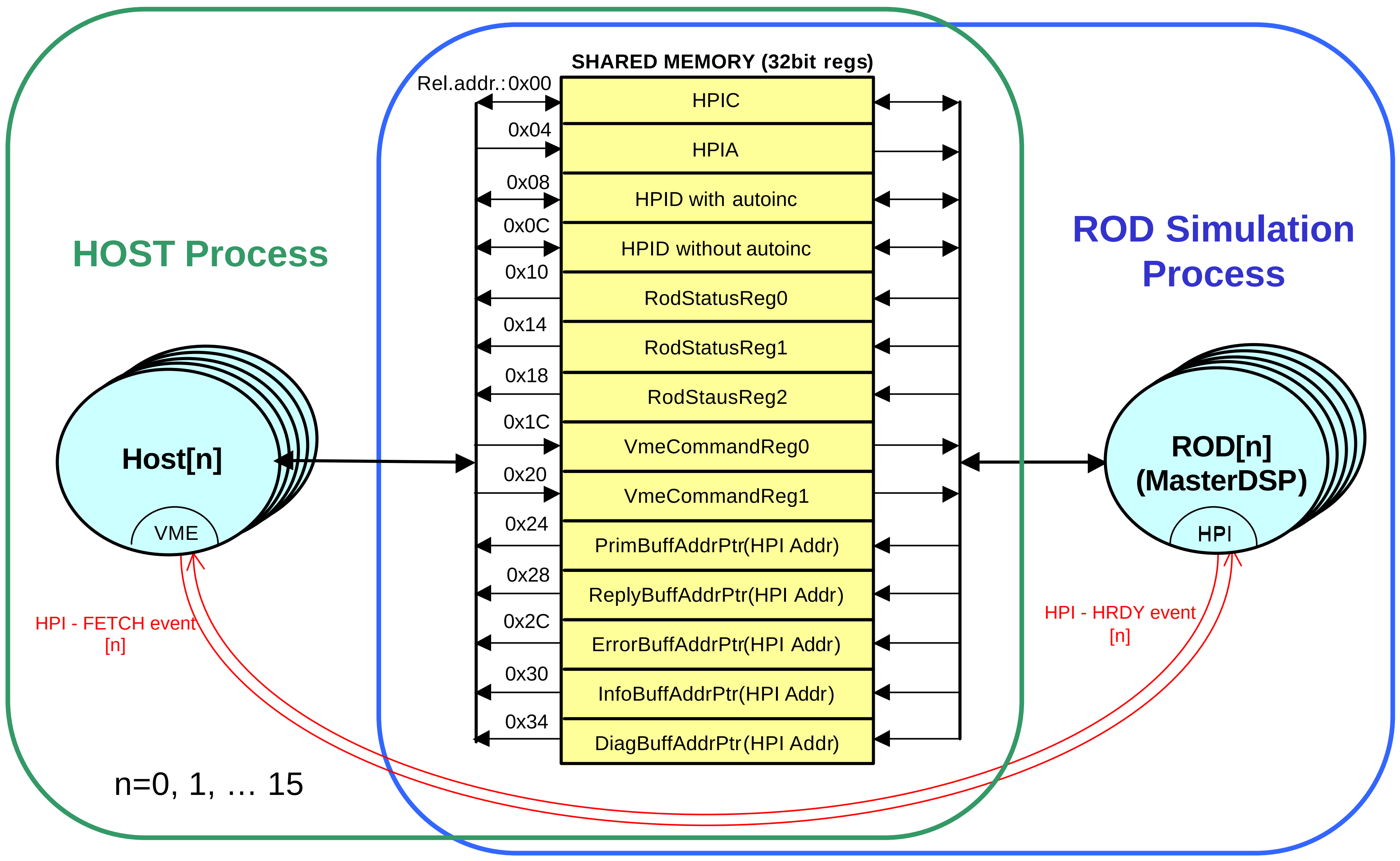
- process structure, threads,
- error handling,
- basic user interface,
- communication with the rods,
- general primitive implementation,
- FPGA "management" (flash memories, status and command registers communication),

• Basic functionality of the program successfully tested :

- Host communication with EVM DSP(PC module) running real MasterDSP code,
- behavior in the “multirod” environment with Rods simulated by the software on PC.

-> Since both the Host program and MasterDSP almost doesn't know, if they talk to the virtual or real counterpart, we don't expect major software problems in the Host-MasterDSP communication in the real word.

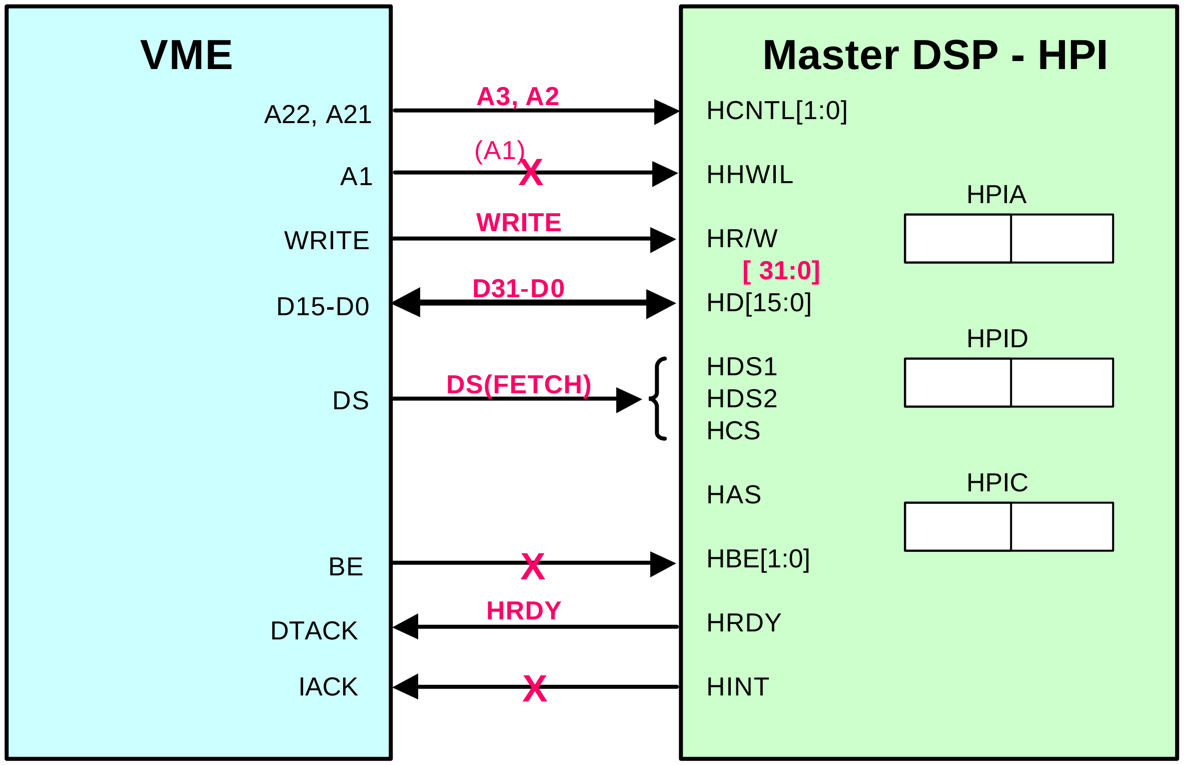
Interface HOST - ROD simulation



Pseudo "VME" address = sharedMemoryStartAddr + rodNumber(n) * SHARED_MEM_SIZE_PER_ROD(0x38)+ relativeRegAddr (0x00 - 0x34);

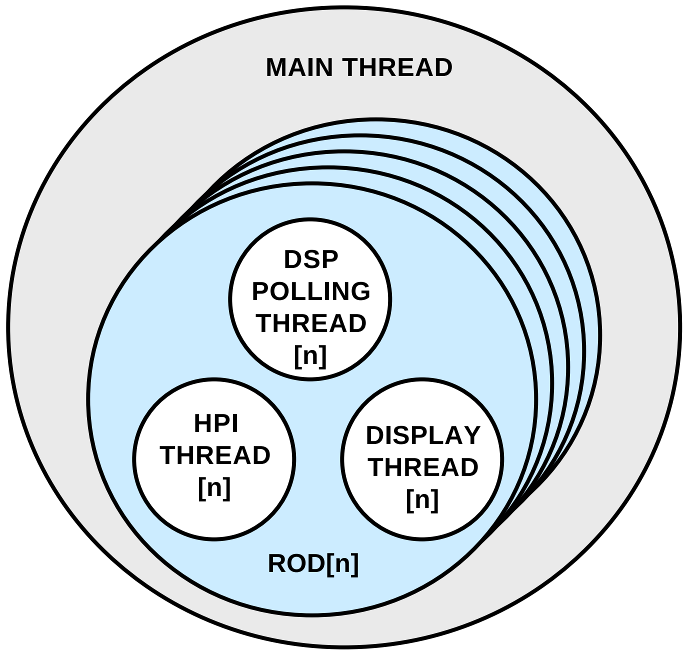
TextBuffsStruct HPI addresses (MasterDSP memory space) are read after Rod Initialization from ReplyBuffer, like in reality.

Interface VME - HPI (ROD simulation)



Simulation:
WRITE signal - bit in HPIC (this bit doesn't exist in the real HPIC);
HRDY signal - Win32 HRDYevent + HRDY bit in HPIC;
DATA STROBE(FETCH) signal - Win32 FETCHevent + FETCH bit in HPIC.

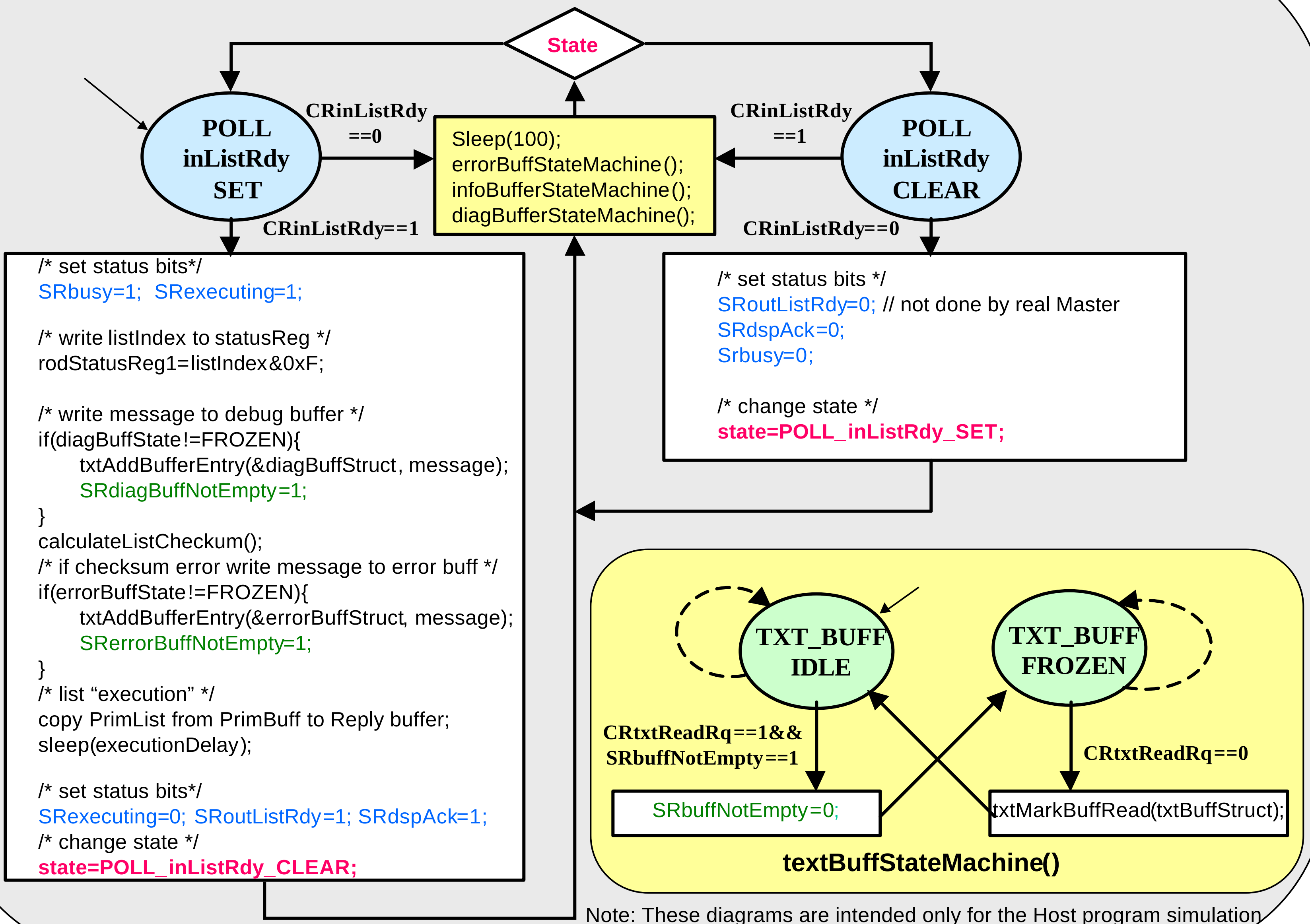
ROD Simulation Process - structure



n = 0,1…(15)

The simulated MasterDSP knows just one primitive - ECHO.
This is absolutely sufficient for the Host testing, since we only need to send any valid PrimList to Rod and read the valid ReplyList back (primitiveList decoding and its “real” execution can be tested directly with EVM DSP running MasterDSP code).
Passing the error and diagnostic messages is also implemented in this simulation.

DSP Polling Loop Thread (only for Host simulation)



```
Sleep(100);
errorBuffStateMachine();
infoBufferStateMachine();
diagBufferStateMachine();
```

```
/* set status bits*/
SRbusy=1;  SRexecuting=1;

/* write listIndex to statusReg */
rodStatusReg1=listIndex&0xF;

/* write message to debug buffer */
if(diagBuffState!=FROZEN){
      txtAddBufferEntry(&diagBuffStruct, message);
      SRdiagBuffNotEmpty=1;
}
calculateListCheckum();
/* if checksum error write message to error buff */
if(errorBuffState!=FROZEN){
      txtAddBufferEntry(&errorBuffStruct, message);
      SRerrorBuffNotEmpty=1;
}
/* list "execution" */
copy PrimList from PrimBuff to Reply buffer;
sleep(executionDelay);

/* set status bits*/
SRexecuting=0; SRoutListRdy=1; SRdspAck=1;
/* change state */
state=POLL_inListRdy_CLEAR;
```

```
/* set status bits */
SRoutListRdy=0; // not done by real Master
SRdspAck=0;
Srbusy=0;

/* change state */
state=POLL_inListRdy_SET;
```

```
SRbuffNotEmpty=0;
```

```
txtMarkBuffRead(txtBuffStruct);
```

Note: These diagrams are intended only for the Host program simulation and don't describe the real structure of the MasterDSP code!!

Test Stand software development plans

To do for the initial testing in September:

- implement all necessary primitives,
- complete the Rod initialization routines,
- complete the user interface,
- improve the error handling,
- continue testing, code "cleaning” and tuning (some small design changes possible) ...

Plans from today to the end 2000

- preparation for the system test, include the TIM and BOC management into the program, add more primitives needed for the system test;
- detailed preparation for a simple DAQ (refer to ***John Hill's talk "SCT ROD Crate DAQ***");

• In general simply:
  - **testing, testing, testing** (V.I.Lenin),
  - continuous improvement of the software, adding new primitives, program features, implement "user" desires and recommendations.

The more advanced and more "final" version of the Test Stand software ready for the user evaluation certainly should be available by the end of November after the first system tests.

• **the exact schedule - refer to *Richard Jared’s “ROD Schedule”* presentation.**